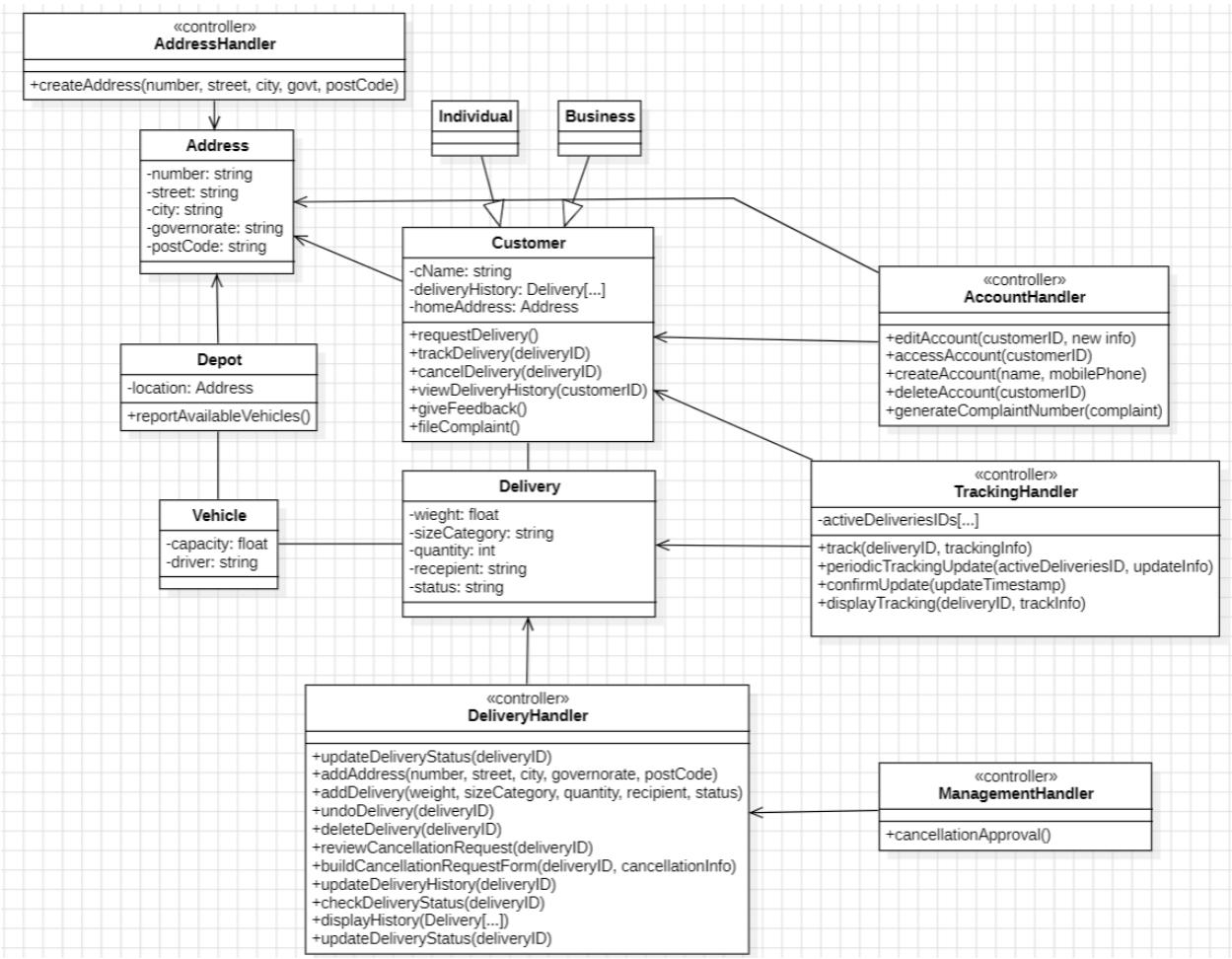
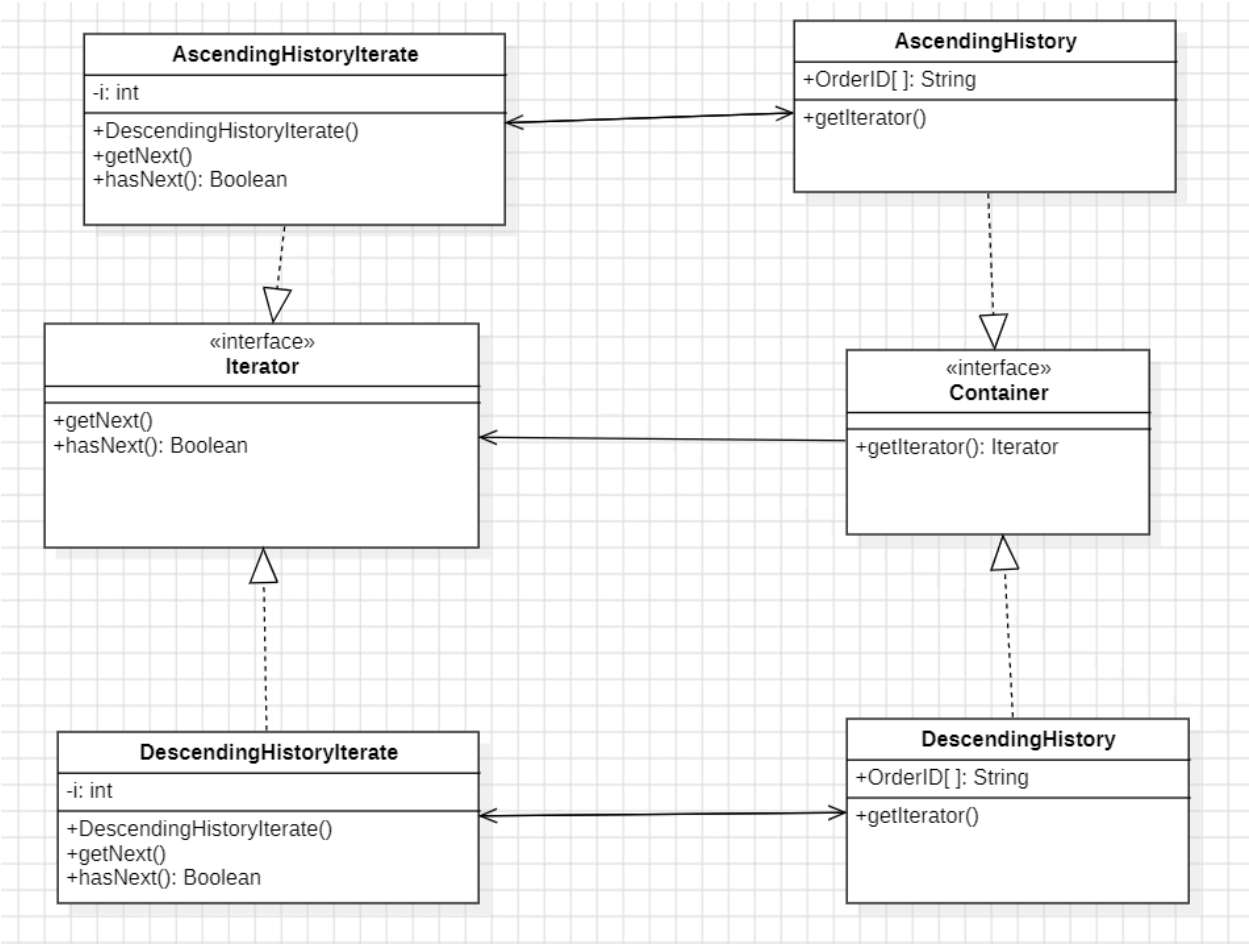
Design Class Diagram



Design Patterns

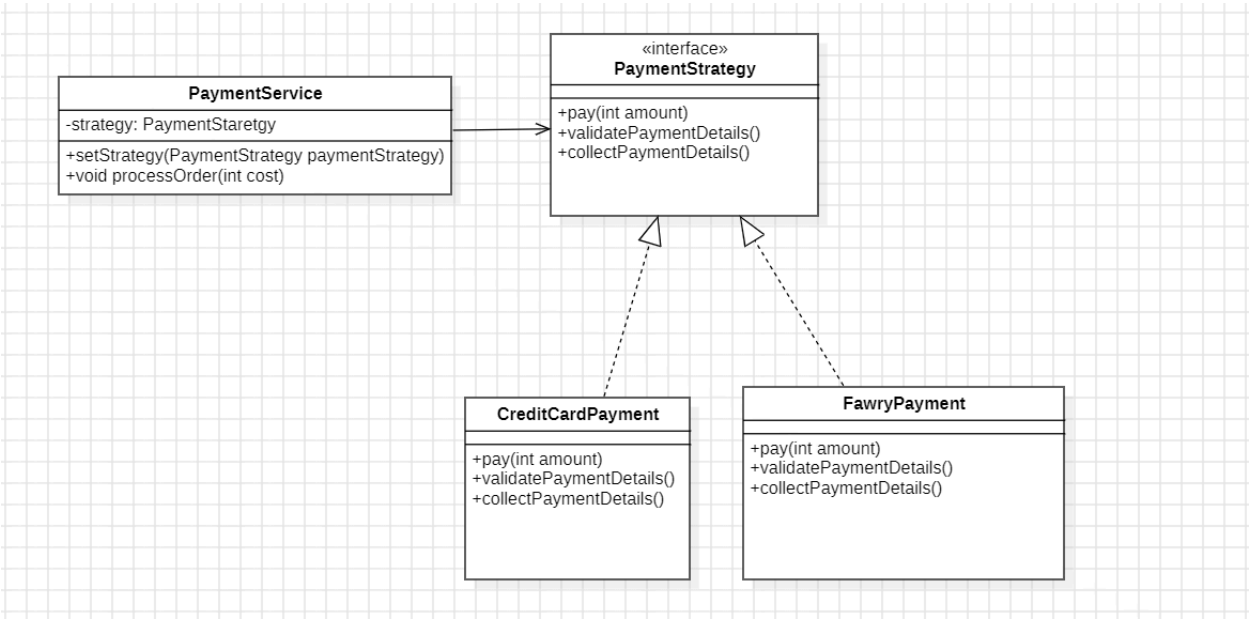
Iterator Pattern

The core idea of the iterator pattern is to provide the means to access elements of a certain collection without exposing the underlying structure of the collection and also allows the implementation of multiple iteration algorithms , making it suitable to implement in the OrderHistory class , where customers would be provided with the option to display their history oldest-new(Descending order) or new-old (Ascending order) without having to repeat the implementation of each algorithm.



Strategy Pattern

The strategy pattern enables selecting an algorithm's behavior at runtime. This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.
The strategy pattern allows the algorithm to vary independently from the clients that use it, making it suitable in implementing the Payment class.
Customers can choose different payment options but in the end 3 things will happen each time , customer will type in their payment details , details will be validated and finally the payment process so instead of repeating this process multiple times we make use of the strategy pattern to tailor each payment option requirements

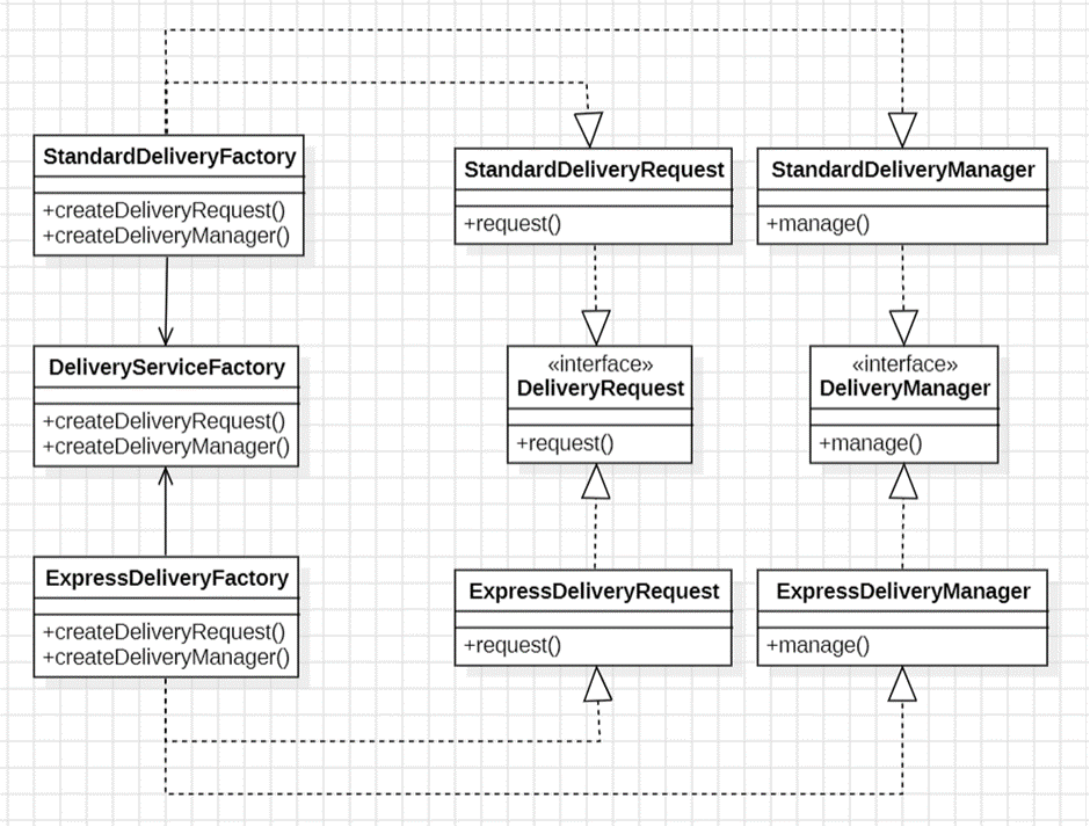


Abstract Factory Pattern

The Abstract Factory design pattern, a creational pattern that provides a way for creating and organizing groups of related objects, is used for creating two entities in the delivery service system, the Standard and Express delivery services. Each entity is responsible for producing two main products/functionalities for the delivery service, which are the DeliveryRequest and the DeliveryManager. Through the DeliveryRequest interface, all customer delivery requests are handled, regardless of whether they are standard or express. Meanwhile, the DeliveryManager interface oversees each delivery by assigning the necessary resources for fulfilling it successfully. By isolating the concrete classes, this design pattern makes it possible to select between different product families, like the standard and express groups. Additionally, it ensures consistency among the products of each group, resulting in easy maintenance and updates that may include adding more products to the delivery system if needed.

Participants:

- Abstract Factory: DeliveryServiceFactory
- Concrete Factories: StandardDeliveryFactory, and ExpressDeliveryFactory
- Abstract Products: DeliveryRequest, and DeliveryManager
- Concrete Products: StandardDeliveryRequest, StandardDeliveryManager, ExpressDeliveryRequest, and ExpressDeliveryManager



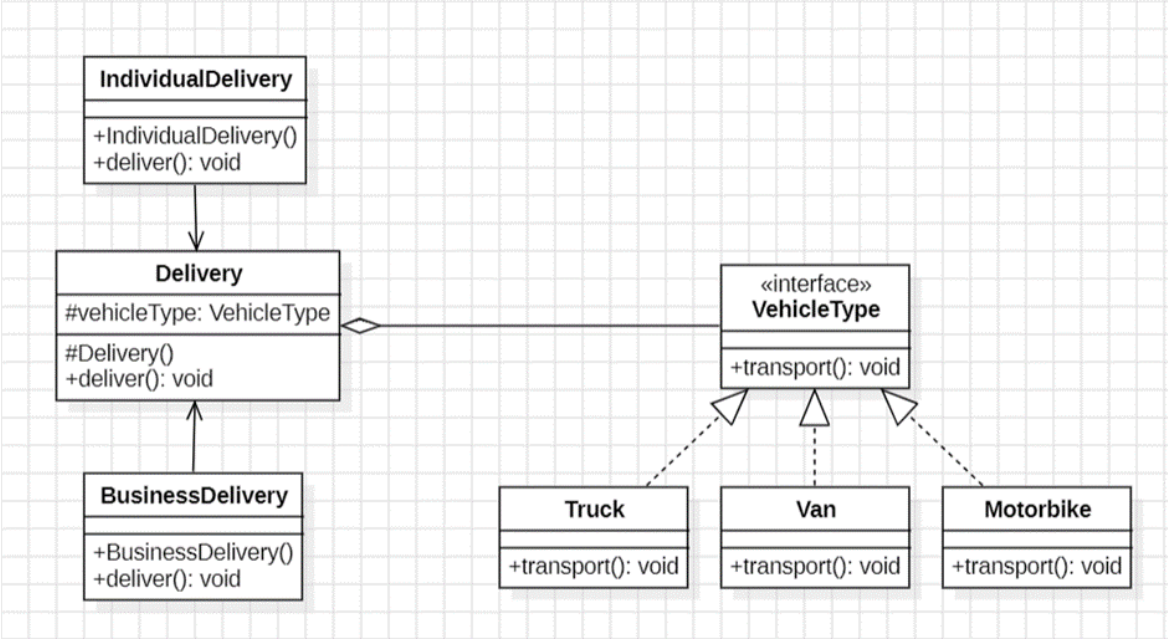
Bridge Pattern

For a system that involves assigning each delivery with a vehicle type, the ideal design pattern is the Bridge design pattern. This pattern effectively decouples the abstraction from the implementation, allowing for the separation of the vehicle assignment from the client view. The Delivery class is divided into two entities, one for individual customers and one for businesses, where it handles the delivery part for each entity. The VehicleType interface, which handles the implementation, has multiple concrete implementations for each type of vehicle that can be assigned to transport a delivery. This design applies the Open-Closed principle, allowing the system to easily be extended with new vehicle types without modifying the existing code. It also follows the Single Responsibility principle, as each class or interface has a defined

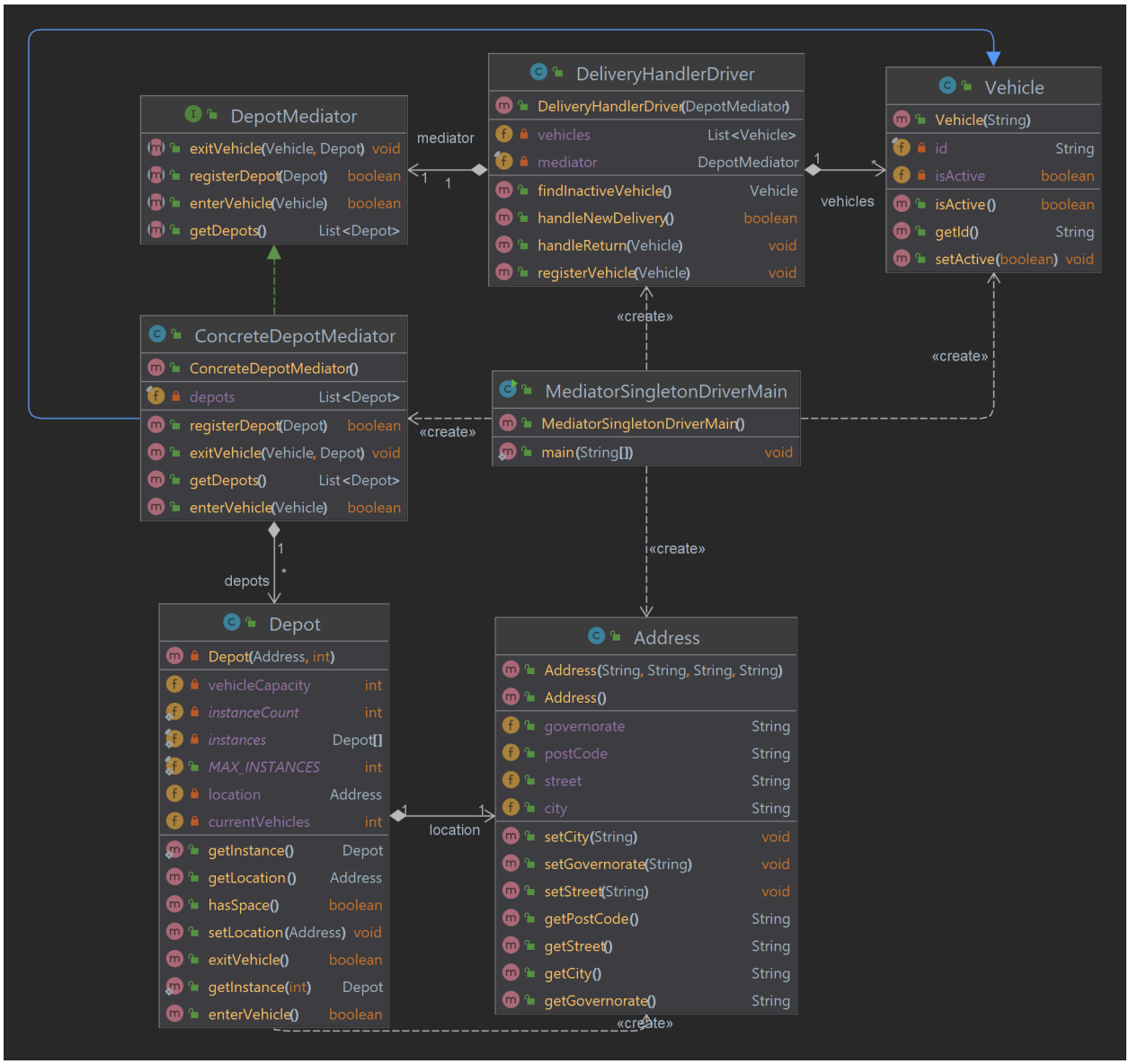
responsibility, where the Delivery class focuses on customer-based processes, while the VehicleType interface manages the details of vehicle assignment.

Participants:

- Abstraction: Delivery
- Refined Abstractions: IndividualDelivery, and Business Delivery
- Implementation: VehicleType
- Concrete Implementations: Truck, Van, and Motorbike



Mediator & Singleton Patterns



Singleton Pattern

Assumptions:

- A maximum of two `Depot` instances are required and sufficient for the application's needs.
- Each depot has a finite vehicle capacity that must be managed efficiently.
- The 2 different instances are called correctly as needed by the caller, meaning when needing the second depot, we call the second instance and not the first.

Purpose:

The Singleton pattern ensures that a maximum of two depot instances are created and managed. This is critical for maintaining control over depot creation and managing vehicle capacity across depots.

Approach:

- **Depot Class**: Manages a depot with a specific location and vehicle capacity.
- **Singleton Logic**: Utilizes double-checked locking (using synchronized() method) to ensure thread-safe creation of up to two depot instances, with various optimizations already applied to the implementation.

Mediator Pattern

Assumptions:

- Depots and vehicles are correctly registered with the mediator to enable proper functionality, there is no vehicle/depot that does not exist within the system, but exists physically.
- Only the mediator organizes and allows entry/exit of vehicles, and no other entity.

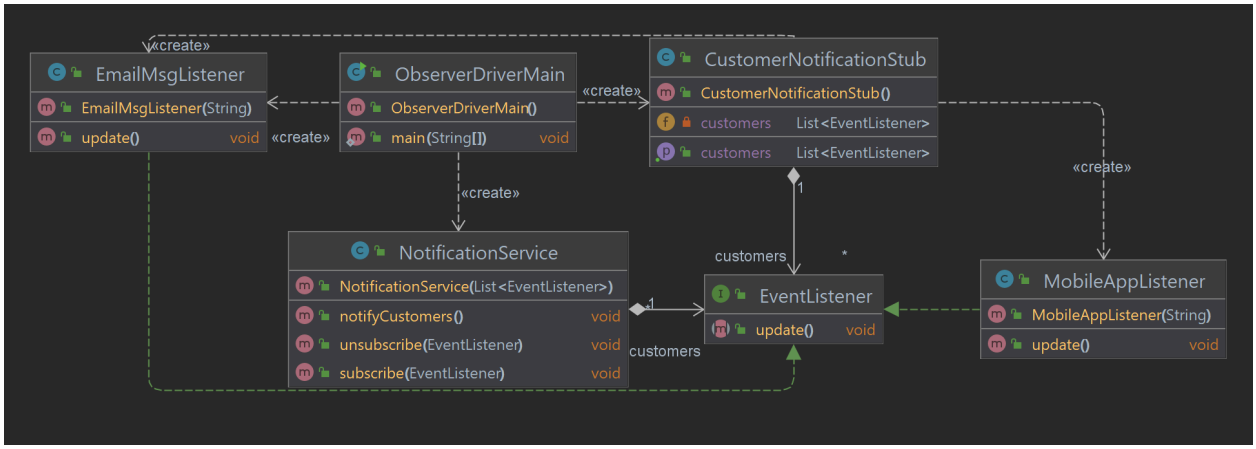
Purpose:

The Mediator pattern organizes and mediates interactions between depots and vehicles, ensuring that vehicles are directed to depots with available space and managing vehicle entry and exit efficiently and without clashes or conflicts among them.

Approach:

- **DepotMediator Interface**: Defines methods for registering depots, managing vehicle entry and exit, and retrieving depot information.
- **ConcreteDepotMediator Class**: Implements the mediator interface, manages the list of depots, and handles vehicle movements.

Observer Pattern



Assumptions

For the observer design pattern, we make the main assumption that we have an existing database of customers on different platforms that can each have different subscription preferences. This builds the base on which the notification service operates.

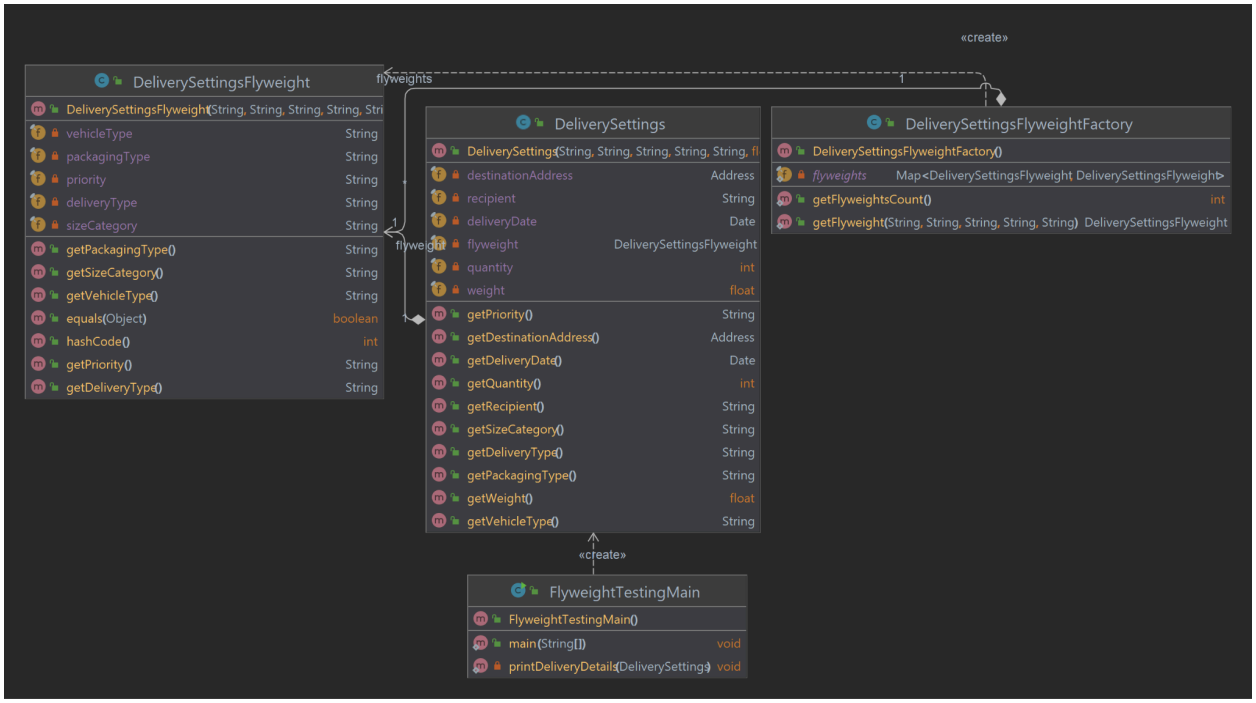
Purpose

The purpose of the observer design pattern here being applied to the notification service class is to facilitate the action of informing the subscribers (customers) of any changes that occur to the publisher (our application). This is in addition to the ease of adding different subscription preferences later on or different kinds of notifications. An example of this would be if we decide to add events, or perks for some users we send out notifications in the preferred channel for each customer.

Approach

In our implementation, the `ObserverDriverMain` creates an instance of the `NotifcationService` object. The notification service makes use of the interface, `EventListener`, since all its attributes and methods are made to deal with it, instead of a specific type of listener. This allows for easy adjustments as the application grows and more functionality is required. Both the email and mobile listeners implement this general listener type and are in charge of actually doing the action of sending the notification to a customer in their preferred channel. This action is then called using the iterator in the notification service class.

Flyweight Pattern



Assumptions:

- **High Object Creation**: The `DeliverySettings` class is expected to create numerous objects, making the Flyweight pattern optimal for memory efficiency.
- **Redundant Attributes**: There are many redundant values among the selected in the intrinsic attributes, making them suitable for separation from the extrinsic ones.
- **Template Usage**: Users may store and reuse "Template settings" or last used delivery settings, further justifying the Flyweight pattern.

Purpose:

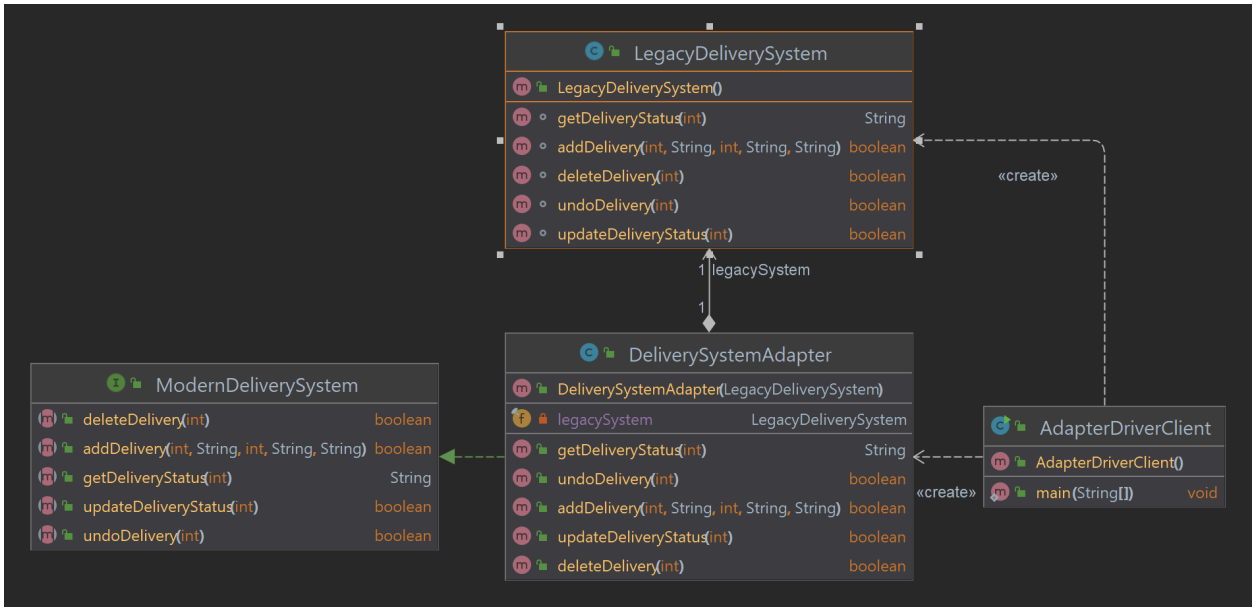
The `DeliverySettings` class is optimized using the Flyweight pattern to reduce memory consumption by sharing common data among its instances. This is particularly beneficial given the high number of delivery objects and the presence of redundant attribute values.

Approach:

- **Intrinsic Attributes**: Attributes that remain constant across multiple instances (e.g., `deliveryType`, `packagingType`, `priority`, `vehicleType`, `status`) are moved to the flyweight class.

- **Extrinsic Attributes**: Attributes specific to each delivery instance (e.g., `weight`, `quantity`, `sizeCategory`, `recipient`, `destinationAddress`, `deliveryDate`) remain inside the original class.
- **Factory Pattern**: A factory ensures that new flyweight objects are created only if an object with the same intrinsic attributes does not already exist, thereby saving memory.

Adapter Pattern



Assumptions:

- The legacy system is fully functional before integrating the adapter.
- The modern system, with its updated functionalities, can communicate effectively with other components as the legacy system did.

Purpose:

The `DeliverySystemAdapter` serves to migrate the legacy delivery system to a more modern version, allowing for parallel deployment. It facilitates this transition by adapting certain functionalities while retaining unchanged methods from the `LegacyDeliverySystem`.

Approach:

- **Functionality Migration**: The adapter provides new implementations for some methods (e.g., `addDelivery`, `getDeliveryStatus`, `updateDeliveryStatus`) while directly using existing legacy methods for others (e.g., `undoDelivery`, `deleteDelivery`).

- **Parallel Deployment**: The adapter enables the modern system to run alongside the legacy system, leveraging the existing infrastructure while gradually transitioning to improved functionalities.
- **Open-Closed Principle**: The design follows the open-closed principle, allowing new features or changes to be integrated smoothly by extending the adapter class without modifying existing code.